The Pennsylvania State University

The Graduate School

College of Engineering

POWER EFFICIENCY AND SCALING OF THE CELL BROADBAND ENGINE

A Thesis in

Computer Science and Engineering

by

Jacob Johnson

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2009

The thesis of Jacob Johnson was reviewed and approved[1] by the following:

Padma Raghavan
Professor of Computer Science and Engineering
Thesis Advisor

Sanjukta Bhowmick
Associate Professor of Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

[1]Signatures are on file in the Graduate School.

Abstract

Since the 1980s, frequency scaling, brought on by Moore's Law, has given us increased uniprocessor performance at a steady rate and with no cost to application programmers. Recently, however, the limitations of interconnect delay and thermal capacity in the hardware have ended this trend, and frequency scaling has become less viable. Computer architects have turned to the chip multiprocessor (CMP) design paradigm to continue the performance trend, but designing and programming these systems has proven to be difficult.

Although general-purpose CMPs have been commercially available for several years, most have had homogeneous designs, and have been used to run multiple independent threads concurrently. The Cell/B.E., on the other hand, is a heterogeneous shared-memory CMP available today, most notably in Sony's PlayStation 3 game console. This thesis will evaluate the Cell/B.E. with respect to the design goals of CMPs.

In addition to increased performance, another motivating factor in CMP development is power efficiency. We will see that, for a variety of applications, porting code to the Cell processor can lead to an energy savings of more than 60% over the same applications on a traditional uniprocessor. We will also see the limitations of using the Cell processor: the difficulty involved in programming on it, and the limitations of the hardware.

In CMPs, memory bus width can become as serious a problem as memory latency. With multiple processing cores issuing memory requests simultaneously, bus contention can lead

to slowdown even when memory latency would be completely masked. This thesis will therefore also address the limitations of scientific computing on the Cell/B.E. with respect to memory bandwidth, and show how this bandwidth can diminish or even eliminate the beneficial effects of increased concurrency.

Table of Contents

List of Tables

List of Figures

Acknowledgements

First and foremost, I would like to thank Dr. Padma Raghavan, who encouraged me every step of the way both as an advisor and a role model. I would like to thank Dr. Bhowmick for her support as well, and Dr. Shontz and all my other professors who have never ceased to amaze me with their knowledge, dedication, and compassion

I would also like to thank my parents and my family for the unquestioning concern and support, and all my fellows, especially Dr. Konrad Malkowski, for doing everything they can to create an environment conducive to personal growth.

We gratefully acknowledge partial support of this work through MPO contract #H98230-07-C-0426.

1 Introduction

Limitations in hardware technology are preventing clock speeds and conventional instruction-level parallelism from increasing even as transistor sizes continue to scale according to Moore's Law [2, 4]. At the same time, the cost of powering and cooling supercomputers is rising as the power use of their component processors increases, and the environmental impact of high-performance computing continues to worsen [27]. Chip designers have turned to chip multiprocessors (CMPs) to continue increasing instructions per cycle (IPC), permitting greater processor performance, and reduce power consumption (see Appendix A) [25, 22].

CMP designs vary greatly between architectural families, but can generally be divided into two categories [30, 29]:

Small-scale CMPs are extensions of the uniprocessor design. These processors tend to be homogeneous, and usually feature distributed caches and a few, powerful, out-of-order cores.

Large-scale CMPs have cores specifically designed for multiprocessing. They feature many, usually in-order, cores, may be homogeneous or heterogeneous, and are tightly coupled [25].

The continuing emergence and proliferation of CMPs is creating challenges in all areas of computing. Algorithms and software must be rewritten, and often redesigned, to exploit

parallelism [28]; compilers must adjust to new ISAs and make hardware management easy for the user [9]; and hardware must continue to offer performance gains while making hardware management easy for the user *and* the compiler [13]. The interaction of new technologies is difficult to predict, and it is often left to the programmer to discover what is and isn't possible with the available tools.

It is hard to predict how CMP technology will evolve, as chip manufacturers are trying a variety of approaches to CMP design. NVIDIA's Tesla architecture, intended for scientific computing, derives from a line of GPUs and features a heterogeneous hybrid GPU/CPU model roughly similar to the Cell architecture [8]. Sun's UltraSPARC T1 offers 32-way simultaneous multithreading across eight homogeneous cores [17]. Intel continues to develop small-scale CMPs, such as their Core 2 line, but is also investigating large-scale multiprocessing with the Teraflops Research Chip, an experimental 80-core architecture designed for scientific computing [7].

This thesis takes scientific computing performance as the motivating factor for this work. Scientific computing applications have a narrower focus than general purpose applications: scientific computing problems have data-intensive numerical solutions. The tools of scientific computing include matrix theory and differential equations, and vectors, matrices, and arrays of arbitrary dimensionality compose the data sets. These problems are the most popular in the world of concurrent computing, as they are often the easiest to parallelize, and have the greatest rewards for doing so [12].

Little's Law, a queuing theory law discovered by John Little, can be applied to multicore architectures and the question of how much bandwidth is required to guarantee a steady stream of data is available to all processing elements. Little's Law states that in a queuing process,

$$L = \lambda W,$$

for L, the expected number of units in the system, λ, the units' expected arrival rate, and W, the expected time a unit spends in the system [23].

We can reshape Little's Law for the specific queuing process of a memory subsystem. In this application, the three variables of the law take specific form:

L: The expected number of concurrent, outstanding memory requests,

$\boldsymbol{\lambda}$: The arrival rate of memory requests, and

W: The time to complete a DMA request; in other words, the memory latency

For the processor using the memory controller, W, the memory latency, is constant as long as requests don't conflict. λ, however, can vary depending on the software and the architecture of the processor. If the application is data-intensive and has poor data reuse, the arrival rate will increase, and the number of concurrent outstanding memory requests will increase. In CMPs, λ can also be affected by the degree of parallelism in the hardware. Little's Law therefore verifies the intuitive idea that increasing the degree of parallelism under a memory subsystem while maintaining the capabilities of each PE will increase the demand for memory. The real effect of this can be seen in our experiments, where λ is increased, driving L to the limits of the hardware.

We will examine specifically the Cell Broadband Engine Architecture (CBEA), a large-scale CMP architecture. This thesis seeks to determine whether the CBEA is an appropriate processor for scientific computing. In particular, we will examine methods for optimization on the processor, its power efficiency, and how memory bandwidth sustains scalability. We will show that appropriate utilization of the processor can result in a great increase in power efficiency when compared to a traditional processor, but that off-chip memory bandwidth can be a significant bottleneck for scientific computing applications.

The rest of the thesis is organized as follows. Chapter 2 examines work related to

off-chip memory in CMPs. Chapter 3 will explain the architecture of the Cell Broadband Engine, and Chapter 4 will examine the challenge of programming on the Cell/B.E. Chapter 5 will describe the setup used for experiments in this work, and Chapter 6 will give their results. Finally, Chapter 7 will end the thesis with concluding remarks and a look at future work.

2 Previous Work

Jiménez-González et al. benchmarked the memory bandwidth of a 2.1 GHz Cell blade in a computation-free application [18]. They were able to achieve a transfer rate of 20 GB/s for read- and write-only applications, and 23 GB/s for read-write, out of a theoretical maximum bandwidth of 23.8 GB/s on that system. The benchmarks they presented in that paper were used to benchmark the Cell system we used; the results of these tests are shown in chapter 5.

Bienia, Kumar, Singh, and Li developed PARSEC, a shared-memory-CMP benchmark [3]. In their paper, they note that increasing the degree of parallelism on a chip will increase demand on the memory controller, and that architectural improvements are necessary to permit full exploitation of parallelism.

Much work has been done studying different types of on-chip memory systems for CMPs [21, 1, 24]. CMPs can have coherent caches, local stores, and shared and distributed L2s and MICs, among many other design choices. These choices can greatly affect performance, but for the sake of this thesis we are ignoring the effect of changing the specifics of on-chip memory.

Zhao et al. studied a large-scale CMP and concluded that an extensive memory hierarchy and high off-chip memory bandwidth would be needed to supply the compute nodes with data [30]. In a constraints-aware simulation of a 32-node CMP, they showed at least 64 GB/s of memory bandwidth would be required, and that a hierarchical cache scheme

with large local L2 caches and a global L3 cache would significantly offset off-chip memory demands. This work shows how memory throughput is a critical factor in CMP performance, and suggested an empirically-derived target for memory bandwidth in a theoretical environment.

Very little work has studied the paradigm shift in off-chip memory design caused by the emergence of CMPs. While work is constantly being done to increase the throughput of the MIC, no study has comprehensively shown how parallelism and the memory subsystem correlate.

3 The Cell/B.E. Architecture

The Cell Broadband Engine Architecture is a heterogeneous, distributed-memory multi-core processor architecture jointly developed by Sony, Toshiba, and IBM. It features one master Power Processing Element (PPE) and multiple subordinate Synergistic Processing Elements (SPEs)—streaming vector processors with no direct access to main memory and limited instruction pipelines.

The Cell/B.E. system used was the first-generation 90nm 80GB Sony PlayStation 3 running Fedora Core release 6. Because it is a commercial package, we had to accept some hardware, such as the GPU and Blu-ray drive, that contribute to the total system power but have no use in these experiments. The system consumes approximately 197 watts while idling; during computation, power consumption reaches 200-220 watts depending on SPE utilization. The Cell processor used in the PlayStation 3 is rated for a peak performance of 179.2 GFLOPS.

Fitting nine processing cores into the same space as a single traditional processor required the sacrifice of some architectural features. Both the PPE and the SPEs feature dual-issue, in-order instruction pipelines, reducing transistor counts by limiting functional unit duplication and reorder hardware.

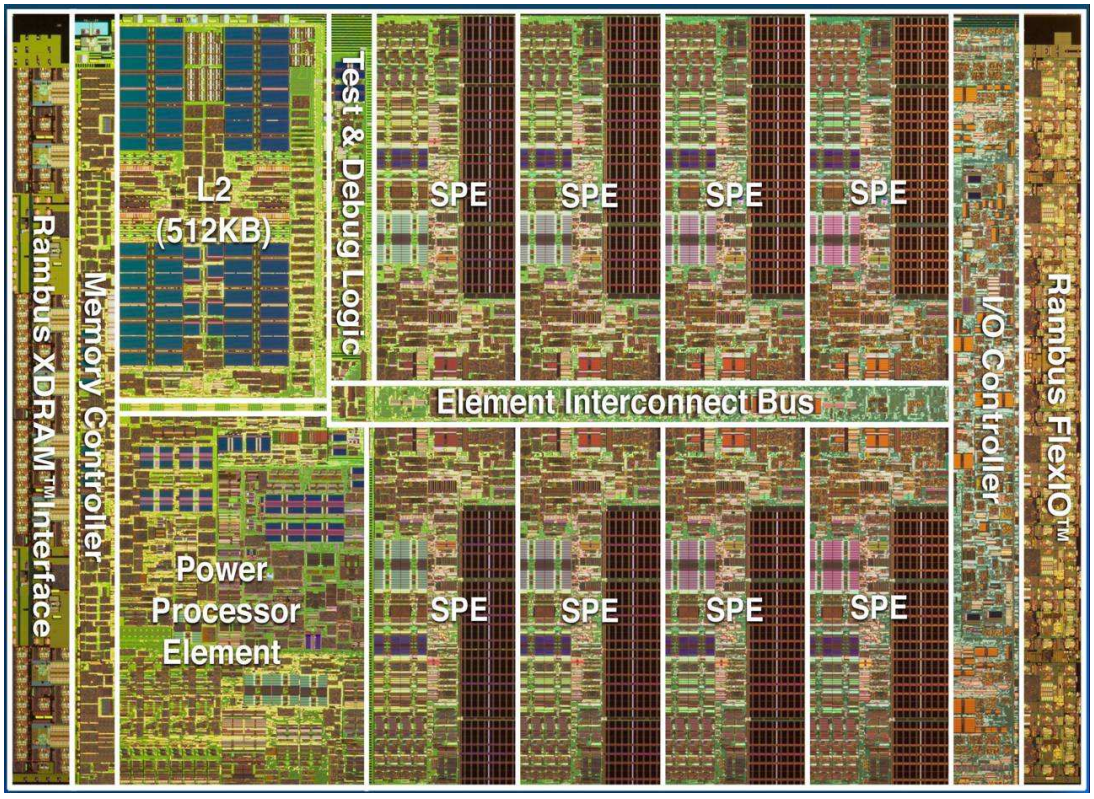


Figure 3.1: Micrograph of the Cell processor [6]

3.1 The Power Processing Element

The PPE is a 64-bit SIMD processor, fully compatible with IBM's POWER ISA with the Vector/SIMD Multimedia Extension [19]. To fit in the transistor budget of the Cell processor, the PPE is a dual-issue in-order processor; the effect of this is that code running on the PPE typically experiences a 2- to 4-fold slowdown versus code running on a comparable processor of the same generation.

The PPE runs the operating system and administers the threads run on the SPEs. Therefore the PPE is not ideal for high-performance computing, even though the its com-

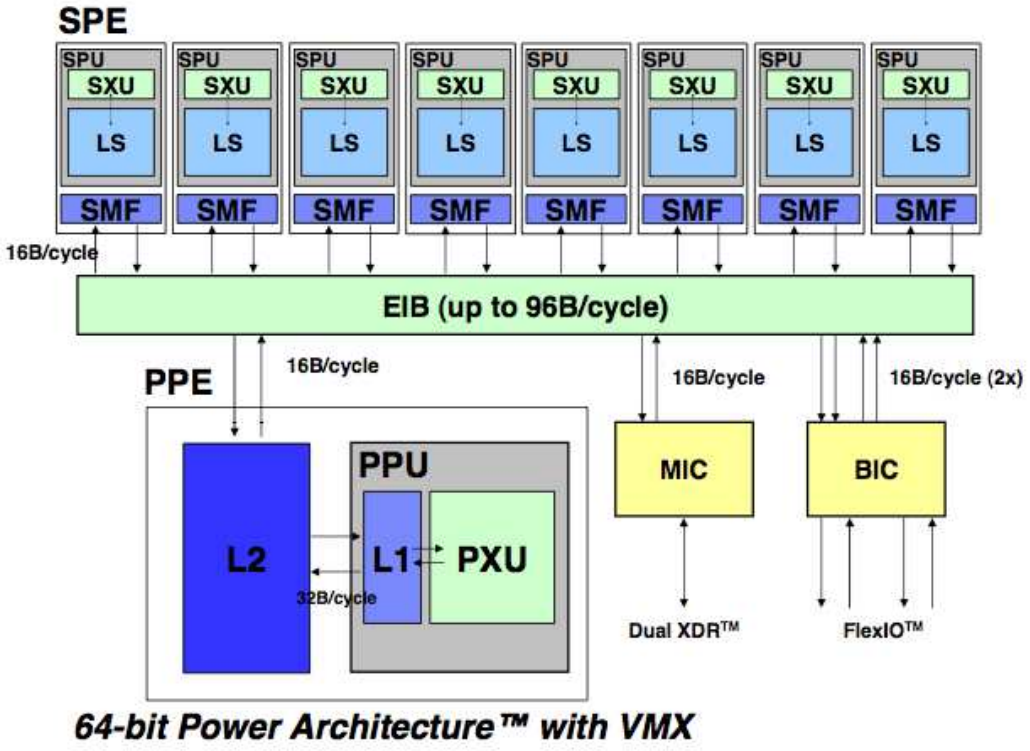


Figure 3.2: The Cell Broadband Engine Architecture [13].

putational power is comparable to that of the SPEs. The improved branch prediction and double-precision performance of the PPE versus the SPEs (see section 3.2) gives it a performance advantage in general-purpose applications, but the isolation of the SPEs gives them advantages in applications that avoid the processor's shortcomings.

3.2 The Synergistic Processing Elements

Each SPE is a 128-bit SIMD processor with its own 256-kilobyte memory space. Cell processors are manufactured to have eight SPEs. In the PlayStation 3, only six are available

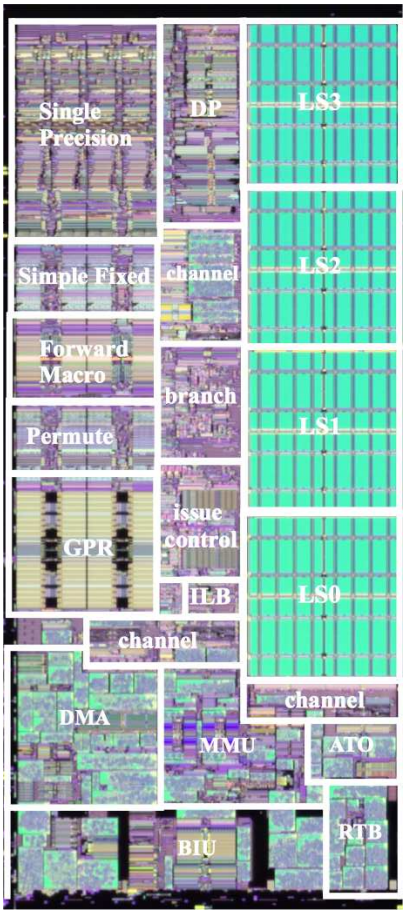


Figure 3.3: Micrograph of a Synergistic Processing Element [11].

to the programmer: one is reserved for the system's hypervisor, and one is disabled to increase manufacturing yield. At 3.2 GHz, each SPE has a peak performance of 25.6 GFLOPS, for a total of 153.6 GFLOPS from the SPEs [5].

Each SPE has a local store of 256 KB of addressable memory which works like a scratchpad memory; main memory is accessible to the SPEs only with DMA operations. Memory management thus becomes the greatest challenge of programming on the Cell processor, and will be discussed further later. SPE program sizes must be limited to fit in

Unit	Instructions	Execution Pipe
Simple Fixed	word arithmetic, logicals, count leading zeros, selects, compares, shifts, and rotates	even
Single Precision	multiply-accumulate	
Byte	population count, absolute sum of differences, byte average, and byte sum	
Permute	quadword shifts, rotates, gathers, and shuffles	odd
Local store	load and store	
Channel	channel read and write	
Branch	branch	

Table 3.1: SPE Instruction Pipes. An odd-pipe instruction can be issued in the same cycle as an even-pipe instruction [10].

the local store, and data must generally be streamed on and off memory.

SPEs cannot support multiple simultaneous program contexts, though the PPE can handle multiple contexts for an SPE. Combined with the lack of caching, this creates a "clean room" environment on the SPEs: program execution is mostly deterministic, as any stalls will be predictable [14].

Because of the drastically reduced size of the SPE (about 14.5 mm^2 and 2-3 watts each [19]) it has limitations not often seen in modern processors. The SPEs have no instruction reordering, speculative execution, branch prediction, or register renaming capabilities [9].

Similar to a VLIW architecture, an SPE can issue two instructions in the same cycle only if they match the dual-issue profile. Generally, an arithmetic operation can be issued the same cycle as a bitwise, memory, or control operation (details in Table 3.1) [10].

SPEs have no branch prediction and an 18-cycle branch penalty [10]. However, branch hint instructions, which direct the processor to prefetch and buffer instructions from the branch target, are available.

Additionally, the first generation Cell processor lacks full support for double-precision floating point arithmetic, which instead uses the single-precision pipeline. As a result, while an SPE is capable of issuing two instructions per cycle, double-precision floating

point instructions can only be issued at a rate of one every seven cycles [19].

3.3 Memory Subsystem

Rambus XDR DRAM provides off-chip memory with a bandwidth of 25.6 GB/s via two channels of 12.8 GB/s RAM. The memory bus is 64 bits wide and fully clocked [10]. Typical memory controller overhead, like refreshing and scrubbing, reduce the peak bandwidth by nearly 1 GB/s. Additionally, the controller uses a bidirectional bus, and the overhead of switching from write-mode to read-mode or vice versa reduces the read-write bandwidth to approximately 21 GB/s [5].

The Element Interconnect Bus (EIB), the on-chip memory bus, is clocked at half the processor speed. The EIB is made up of four rings, two directed clockwise and two directed counterclockwise. Each ring can carry three transfers simultaneously, but these transfers can't overlap. Assuming no collisions, each PE on the EIB–which includes the 6 SPEs, the PPE, the MIC, and the two BICs (Bus Interface Controllers)—can transmit 16 bytes every cycle, for a total peak bandwidth of 256 GB/s. This number is ephemeral, however: sustainable bandwidth tops off at 153.6 GB/s [10, 5] because of architectural limitations. In practice, the EIB bandwidth will be significantly lower than even this figure for a number of reasons. The ring architecture of the EIB means that the relative positions of the source and destination affect latency, and new memory requests can be delayed if every ring is full. Additionally, the bandwidth of the EIB is nearly an order of magnitude greater than the off-chip memory bandwidth, meaning off-chip memory will often be the bottleneck for memory performance [5].

The 512 KB L2 cache services the entire processor, and maintains cache-line coherency [14].

4 Data Staging and Cell/B.E. Optimizations

As previously discussed, the Cell/B.E. architecture presents both great opportunities and great challenges. This section will describe in detail methods for exploiting the performance potential of the Cell. This task can generally be divided into two problems: getting the instructions and data onto the local store of the SPEs (data staging) and getting the program code to execute quickly in the SPU (optimization).

4.1 Data Staging

With only 256 KB of memory available in the local store (LS) of each SPE, and no caching offered by hardware, any high-performance computing application will require explicit memory transfers [19]. Each SPE has a memory flow controller (MFC) that allows direct memory access (DMA) between the local store and main memory [13].

Several options for memory management are available to programmers:

Individual DMA: Data buffers can be transfered directly in chunks as large as 16 KB, and as small as 128 bits, by issuing a command to the MFC. The MFC can be exploited in this way to nearly completely mask memory latency, as will be shown later.

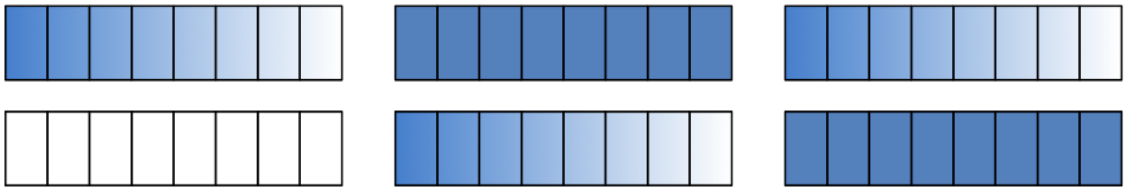
Software caches: Software can manage a cache-like structure in the local store. When memory reads are stochastic, DMA brings in memory lines on demand. The perfor-

mance of a software cache is comparable to that of a lazy hardware cache. If reads have little or no spatial proximity or repetition, each read may result in a DMA request for a single data item, which can slow down the system considerably.

DMA lists: The programmer can assemble a list of up to 2048 DMA transfers in the local store and order the MFC to process these transfers independently. DMA lists can be used to reduce the processing overhead of DMA, and can also replace slower software caches when data accesses are predictable but non-sequential.

Software-level memory management has one clear advantage over hardware caching: if the application programmer is able to predict the order of memory accesses, as is often the case in HPC applications, memory transfers can be heavily or completely masked by computation. Most of the SPE kernels discussed here employ double buffering to hide memory accesses. In this scheme, two buffers are designated in memory for each data construct. While the processor works on the contents of one buffer, the MFC is moving data into or out of the other buffer; Figure 4.1 presents a graphical representation of double buffering.

However, the difficulty of handling memory management makes it a significant problem to application programmers. The program flow and the access patterns of memory must be understood before an approach can even be decided on. In some applications, different streams of data will require different solutions. Once the solution is found, it must be tuned to meet the application profile: for example, quadruple buffering may be preferable to double buffering, or long cache lines may give better results than shorter ones. At all times, the limitation of the 256 KB local store is present; increasing the buffer depth, cache line size, or buffer length will increase the total memory footprint of the SPE program, and usually one or more attribute must be shrunk to make room for another.



(a) When the program begins, the local store is divided into two buffers, and data is transfered into one of them. Computation is stalled until the first complete data chunk is available.

(b) After the first buffer is filled, computation begins on those data items. At the same time, DMA begins filling the second buffer with the next chunk of data elements.

(c) When both buffers are finished, the cycle continues: computation begins on the second buffer while data is transfered into the first buffer. As long as the *computation* on one set of data elements is slower than the *transfer* of one set of data elements, memory latency can be masked.

Figure 4.1: Double buffering

4.2 Optimization

Optimization on the Cell's SPEs mostly involves utilizing their SIMD capabilities and mitigating the shortcomings of the hardware. Many methods commonly seen on traditional architectures, such as loop unrolling, are available (and occasionally more effective) on the SPEs, but won't be included here.

Every operation on the Cell is issued as if it were a vector operation [10]. Vectorization, therefore, provides an opportunity for costless code optimization. The vector instructions support various vector element sizes between 128-by-1b and 2-by-64b [19]. The experiments presented in this thesis all use either 32-bit single-precision vector elements or 64-bit double-precision elements.

The vector intrinsics also provide other opportunities to improve performance. Many common bitwise operations are available in a single vector intrinsic, as well as a combined floating-point multiply-add. Branches can be avoided sometimes with a vector bitwise select [19], a method which we will see in the next chapter.

5 *Experimental Setup*

This section discusses preliminary experiments and the design and function of our benchmarks, which are divided into those intended to test power efficiency and those intended to test scalability.

5.1 Power Benchmarks

Two application benchmarks written for traditional processors were taken to test the Cell processor's power efficiency. We chose two platforms to run and compare the results on: a PlayStation 3 with the Cell processor and an Intel general-purpose CPU. The Intel processor is a 3.2 GHz dual-core Intel Pentium D. Both processors have the same clock speed, and both have approximately the same power draw of over 200 watts when active.

The power draw of the two systems was measured by plugging them into a simple wattage meter. The benchmarks were then run with the same parameters on the different systems and setups, and the meter reported the power use of the systems. Summing the power numbers over the course of a run of each benchmark gave us the total energy, in joules, used by either system to complete the same amount of work.

The two benchmarks used are the Matrix-Vector benchmark and the Factor benchmark.

Program 1 Original Matrix-Vector benchmark C code

```
for(i = 0; i < problem_size; i++)
{
    temp = A[j][i] + Y[IA[j][i]];
    scratch[i] = max(scratch[i], temp);
}
```

5.1.1 Matrix-Vector benchmark

The Matrix-Vector benchmark takes a sparse matrix and a series of vectors as input. It performs an operation similar to matrix-vector multiplication, but instead of summing the products of the array elements, the benchmark keeps the maximum product result. The C code for the innermost loop of the kernel is shown as Program 1, with several trivial changes made for readability. Here `A` is the input matrix, `Y` is an input vector, `IA` is the array of indices of `A`, `scratch` is the output scratchpad, and `i` and `j` are row and column indices. The core operation is a multiplication, but to curtail rounding error the math is done in log space, so the multiplication becomes an addition in code.

Porting this functionality to the Cell processor posed three challenges:

1. The operands were double-precision floating-point numbers, which have slow arithmetic on the Cell.

2. The 'maximum' operation required a conditional operator, putting a branch in the innermost loop of the benchmark kernel.

3. The input matrix was sparse, and so the vector data accesses were not sequential.

The first two issues are addressed by utilizing the SIMD capabilities of the SPEs. Pairs of array elements are put into 128-bit vectors, and SPE vector intrinsics replaced the original `max()` operation with a branchless bitwise one.

Program 2 Vectorized Matrix-Vector benchmark C code

```
for(i = 0; i < BUFFER_SIZE / 2; i++)
{
    /* Compute and insert the first vector element */
    temp = Y[i*4 + IA_old[i*2]%2] + A[i*2];
    v_temp = spu_insert(temp, v_temp, 0);

    /* Compute and insert the second vector element */
    temp = Y[(i*4+2) + IA_old[i*2+1]%2] + A[i*2+1];
    v_temp = spu_insert(temp, v_temp, 1);

    /* Create the vector backup */
    v_backup = spu_insert(temp, v_temp, 1);

    /* SIMD compare-greater-than instrinsic */
    result = spu_cmpgt(v_temp, Scratch_in[i]);
    /* spu_sel puts the vector elements of either Scratch_in[i] or v_backup
     *  into Scratch_out[i] depending on the value stored in result */
    Scratch_out[i] = spu_sel(Scratch_in[i], v_backup, result);
}
```

The third issue presented a greater challenge: using a software cache to access elements of `Y`, the obvious solution, created a significant bottleneck. The better solution was a hybrid of DMA and software cache functionality: the array of indices is used to assemble a DMA list transfer of elements from `Y` on main memory to `Y` in the local store, reordering the elements in their order of access. Because DMA lists aren't designed to transfer single data items, each transfer had to be increased to 128 bytes, or two data elements, adding some complexity to the program. Using this method, we are able to achieve near-perfect transfer rates for non-sequential data items (transfers are nearly as fast as a single sequential DMA).

The vectorized C code for the benchmark is shown as Program 2, again with some slight modifications for readability. The math to select an element in the dense vector `Y` is complicated by the vectorization of the loop as well as some complexity added by using

DMA lists to retrieve elements of `IA`. The apparently redundant `spu_insert()` call avoids a problem with the `spu_cmpgt()` intrinsic, which can change the sign of double-precision floating-point operands.

5.1.2 Factor benchmark

The Factor benchmark is a bitwise integer benchmark. The input and output to the benchmark are streams of N 64-bit integers. The benchmark's operations include factoring and inversion of the input.

Accesses to the vectors are linear and sequential, allowing memory accesses to be masked with DMA transfers. However, the benchmark employed a tradeoff to improve performance at the cost of memory: a 512 KB inverse lookup table, which is too big to fit on the local store of any SPE. We initially used a software cache to access elements of the table individually, but this was too slow. We then tried reversing the tradeoff for which the table was established, sacrificing some computation speed to allow for full-SPE parallelization; this gives us significantly improved performance over the cache. The way this was accomplished in the code is shown in Figure 5.1.

Optimization of the benchmark kernel began with vectorization. At the heart of the benchmark is a population count[1]. The SPE ISA includes a vector population count, which allows us to reduce the number of operations this takes and perform two counts at once. Unlike in the Matrix-Vector benchmark, which used the PPE for synchronization and message passing, in Factor the PPE was free to help with the computation, which provides a small speedup.

The benchmark highlights two of the difficulties caused by using the PlayStation 3 as our Cell platform. The benchmark is not designed to be able to fit in virtual memory; disk

[1]A population count of a data element is simply the number of "1"s in its bit representation. For example, the population count of an unsigned integer 9 (`1001`) is 2.

Original benchmark	Benchmark on the Cell

`for i = 0; i < 0x000FFFF; i += 1`
`  k = i & 0x00000007;`
`  k = (k * (2 - i * k)) & 0x0000003F;`
`  k = (k * (2 - i * k)) & 0x00000FFF;`
`  k = (k * (2 - i * k)) & 0x0000FFFF;`
`  TBL[i] = k`
`endfor`
	`for i = 0; i < 0x0000FFF; i += 1`
	`  k = i & 0x00000007;`
	`  k = (k * (2 - i * k)) & 0x0000003F;`
	`  k = (k * (2 - i * k)) & 0x00000FFF;`
	`  TBL[i] = k`
	`endfor`

`for i = 0; i < n; i += 1`
`  C[i] = TBL[B[i] & 0x00000FFF];`
`  C[i] = (C[i] * (2 - B[i] * C[i]));`
`endfor`
	`for i = 0; i < n; i += 1`
	`  C[i] = TBL[B[i] & 0x00000FFF];`
	`  C[i] = (C[i] * (2 - B[i] * C[i])) &`
	`  0x0000FFFF;`
	`  C[i] = (C[i] * (2 - B[i] * C[i]));`
	`endfor`

Figure 5.1: Changing the lookup table to fit it in the local store. `TBL` is the lookup table, `C` is the output vector, and `B` is the intermediate vector. The size of the table is changed from 65,536 elements to 4096, and the computation of the four uppermost bits is now done on-demand instead of before benchmark execution.

reads and writes are written into the benchmark to load and store the streams. The 5400 RPM hard disk drive used in the PlayStation is significantly slower than most modern hard drives, pushing the bottleneck off of the processor and onto disk I/O speeds. Additionally, the PlayStation 3 has only 256 MB of RAM, of which about 150 MB are generally available to the user. Disk I/O is therefore more frequent on the PlayStation 3 than would be on any modern general purpose system.

5.2 Scaling Benchmarks

Little's Law tells us that the number of units in a system will increase if the arrival rate increases, even when the time in the system stays constant. In single-threaded, uniprocessor systems, the arrival rate is increased by decreasing the time between memory operations—increasing the core clock speed or optimizing code. On a CMP, however, memory requests

can occur concurrently, so increasing the degree of parallelism can increase the arrival rate. To test the effects of memory bandwidth on parallelism, we created parallel applications with various rates and kinds of serial memory accesses.

The scaling benchmarks were run on 1 to 6 SPEs. Ideally the performance should increase linearly: running on 2 SPEs, the benchmark should complete in half the time it took to complete on 1 SPE. By comparing the speed of the benchmark across all numbers of SPEs, we can see how close the benchmark comes to achieving the optimal speedup.

The kinds of memory accesses define the application, and will affect the peak performance. They can be divided into:

Read-only applications, which have have little or no output,

Write-only applications, which have little or no input, and

Read-write applications, which have significant input and output.

Before we tested how the Cell processor handled different types of memory accesses, we had to set a baseline for performance. The theoretical limit on memory bandwidth provided by the hardware is 25.6 GB/s, but achieving this rate is not reasonable. Small factors, such as asymmetry among the SPEs and operating system interference, can have a significant impact when approaching this peak. Additionally, switching from a read-mode operation to write-mode operation on one memory channel incurs a fixed penalty [5].

The benchmark used here was adapted from the one used by Jiménez-González et al. [18]. The benchmark runs on the SPEs and requests memory in chunks of increasing size. 32 memory requests are filed by each SPE without any synchronization to wait for transfers to complete, and the SPE decrementer is used as an accurate timer. With no computation done, and memory requests being filed past the point of saturation, this benchmark should give an accurate picture of the upper limit of the bandwidth provided by hardware.

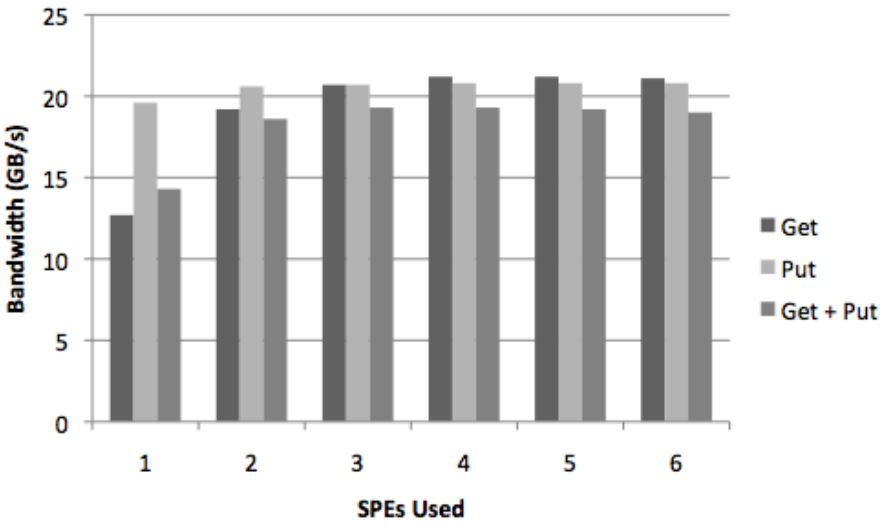


Figure 5.2: Cell memory benchmark results

Figure 5.2 shows the result of benchmarking the memory bandwidth of the Cell processor. A single SPE is only able to issue enough requests to utilize 50-77% of the bandwidth offered by the hardware—about 50% for read and read/write operations, and 77% for write-only operations.

Using all 6 SPEs to issue memory requests, we are able to achieve over 80% of the theoretical maximum bandwidth for read-only and write-only operations, and 74% for read/write operations. Taking the numbers cited in [5], which factor in hardware overhead, as more reasonable theoretical limits, this increases to 85% of the maximum for read- or write-only, and 90% for read/write.

We also see that the performance with only 2 SPEs is consistently greater than 90% of the best performance achieved. We can therefore say that using multiple SPEs allows us to over-saturate the memory controller with requests. The time wasted by the processors waiting for memory requests to be satisfied by the controller can be put to productive use.

We also put the Cell processor to use in three real, dense applications to show that bandwidth limits can affect computation. The benchmark applications include:

Dot product. An input-only benchmark. Data from two vectors is streamed in simultaneously, and the values of corresponding entries are multiplied and added to a global sum. The output is therefore just a single value produced by each processing element.

Pseudorandom number generator. An output-only benchmark. The PRNG used is a linear congruential generator (LCG), in which elements are defined by the recursion: [26]

$$z_{n+1} = a^n z_n + c \bmod m$$

LCGs are sensitive to the choice of values for a, c, and z_0; values from [20] were used to ensure propriety.

Vector addition. An input-output benchmark. Two vectors are streamed in, and an element-wise addition produces an output vector.

While the practicality of each of these functions is limited, they all represent real work; their significance lies in the relative amounts of computation and communication, and not in what that computation achieves.

The optimization of these benchmarks was restricted to simple vectorization and double buffering; instruction fusion, loop unrolling, and compiler optimization weren't used. Because we are measuring speedup across the SPEs, further optimization of the code will not have an effect on the results.

6 Results

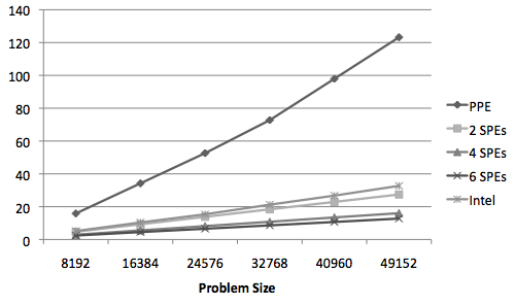
This chapter will report and analyze the power efficiency and scaling of the suite of benchmarks using the methodology presented in Chapter 5.

6.1 Power Efficiency

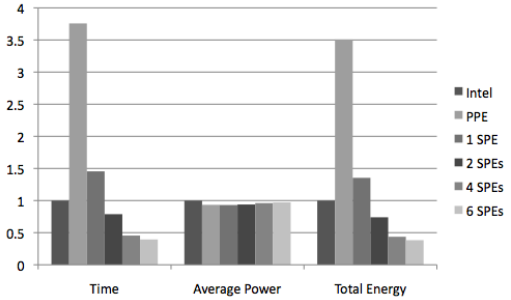
First we consider the total energy use of the Matrix-vector benchmark on our two platforms, the Cell processor and the traditional Intel processor. Total hardware energy is a function of the time to execute the benchmark and the rate of power consumption during execution, so both will be seen.

Figure 6.1(a) shows execution time as problem size is increased across both platforms and with various numbers of SPEs active on the Cell. We see that the benchmark on the x86-64 Intel processor is already nearly four times as fast as the same code, compiled with the same compiler, running on the Cell's PPE.

Figure 6.1(b) shows relative performance for a fixed problem size. The Cell version on 2 SPEs outperforms the Intel version by approximately 25%, despite the poor performance on the Cell's PPE. Using all six SPEs, the Cell processor is 2.6 times as fast as the Intel processor, and 9.5 times as fast as the code on only the PPE. Power consumption is lower on the Cell processor, but not significantly, so the benchmark on 6 SPEs is 2.6 times as energy efficient as on the Intel processor.

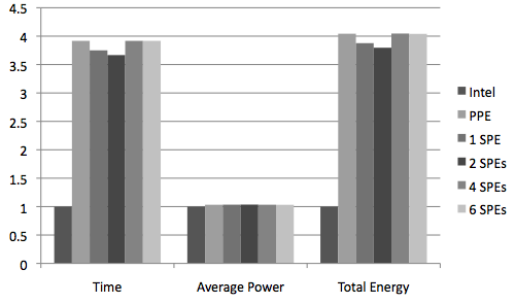


(a) Total execution time of the benchmark for the PPE, SPEs, and Intel processor. We can clearly see that the code running natively on the PPE is already several times slower than on the "comparable" Intel processor.

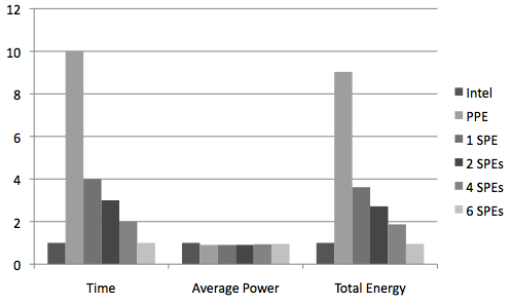


(b) Relative execution time, total energy, and active energy. Power consumption is roughly equal across the board, but running the benchmark with 6 SPEs decreases total energy by decreasing the execution time.

Figure 6.1: Matrix-Vector benchmark results



(a) Relative execution time and average and total power with file I/O. The total energy used on the Cell processor is significantly higher the energy used on the Intel processor, and is even across all fields, because the Cell benchmark is disk-speed bound.



(b) Relative execution time, power, and total energy with no file I/O. Here, the benefit of using SPEs is visible.

Figure 6.2: Factor benchmark results

Benchmark	PPE	1 SPE	2 SPEs	4 SPEs	6 SPEs
Matrix-Vector	350.7%	135.3%	74.0%	43.6%	38.3%
Factor, File I/O	404.2%	387.7%	379.6%	404.6%	404.0%
Factor, No File I/O	903.3%	361.6%	271.7%	187.1%	95.3%

Table 6.1: Power use of the benchmarks on the Cell, as percentage of power used by the Intel benchmark. Total energy is equal to average power times execution time.

Next we will examine the energy efficiency of the Factor benchmark. The benchmark was recorded in two ways: the original format, and with the file I/O operations removed, as explained in Chapter 5. The original benchmark is bottlenecked by file I/O on the Cell, and so is not representative of the processor's performance. Differences in disk I/O speeds result in a consistent slowdown by a factor of 4. Again the power draw of the two systems is nearly identical for the duration of the tests; the total energy consumption therefore mirrors the execution times, with the Cell version of the benchmark using approximately four times as much energy as the Intel one.

We now consider the benchmark with no file I/O operations (results shown in Figure 6.2(b)), and see that the Cell processor just manages to keep up with the Intel processor. This is partly because the benchmark was abandoned when it became clear that file I/O would make the PlayStation 3 noncompetitive in this benchmark. We can see that performance increases with the number of SPEs utilized, with an approximate 10-fold speedup on 6 SPEs versus the PPE, but the increase is not enough.

Table 6.1 shows the hardware energy of the three versions of the benchmarks on the Cell relative to the Intel processor. This is the total energy used by the system to run the benchmark, which is a function of both running time and processor power. We chose systems to have similar power profiles, so total energy use is similar to total execution time.

These results show that reworking an appropriate application to exploit the concurrency of the Cell processor at its low hardware power can result in increased performance per

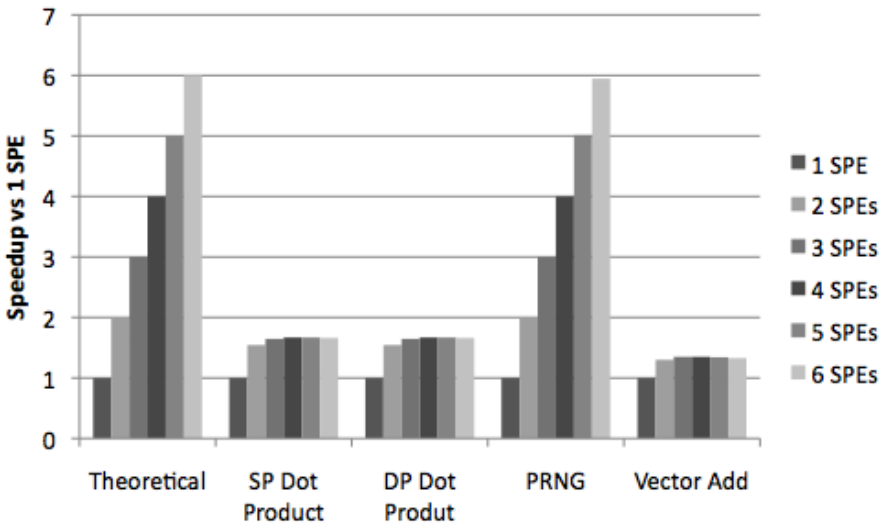


Figure 6.3: The speedup of the four benchmarks on 1-6 SPEs versus 1 SPE. The theoretical peak performance is based on a linear speedup.

joule over a serial version of the application on a traditional processor.

6.2 Scaling

When the degree of hardware concurrency is increased for a parallel application, the performance of the application should improve. This section will show the results of benchmarks designed to demonstrate how the off-chip memory bandwidth of the Cell/B.E. processor helps or hinders this goal.

Both single- and double-precision versions of the dot product benchmark were produced, but the results, shown in Figure 6.3, are the same. In both cases, the vector-optimized computation was able to perform faster than memory could load the data. As the computation is memory-bound, increasing the number of SPEs involved in the computing had no return; they can only compute as fast as the off-chip memory bus can retrieve the data.

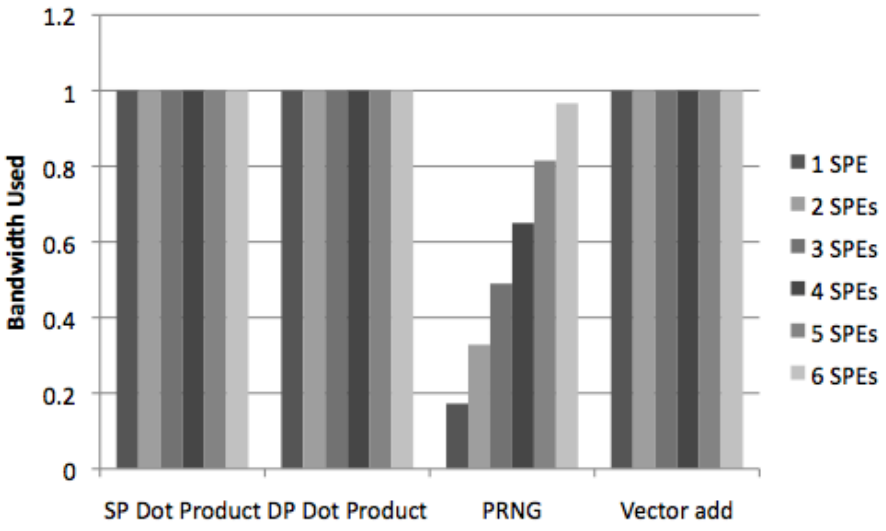


Figure 6.4: Fraction of benchmarked bandwidth used by each benchmark. The dot product and vector add benchmarks both use the full bandwidth for 1-6 SPEs.

On the Cell processor, performance falls well under the theoretical peak because we are limited by memory bandwidth: using all six SPEs, we see only 27.7% of the theoretical speedup, and a 1.7-fold speedup over the single-SPE performance.

The six SPEs offer only 6.3 GFLOPS of double-precision performance (compared to the 153.6 GFLOPS in single-precision), yet the computation is still memory bound. At 25.6 GB/s, the memory bus is able to load 3.2 million 64-bit numbers per second; since each pair of floating-point operations uses two data elements, this translates to a maximum achievable performance of 3.2 GFLOPS across the SPEs.

The pseudo-random number generator benchmark is more compute-intensive than the dot product benchmark. The PRNG is compute-bound, and reaches a nearly linear speedup: 6 SPEs achieve 99% of the linear speedup target.

The vector addition benchmark has significant input and output, but the result is the same as the dot product benchmark: even with few SPEs computing, the processor is

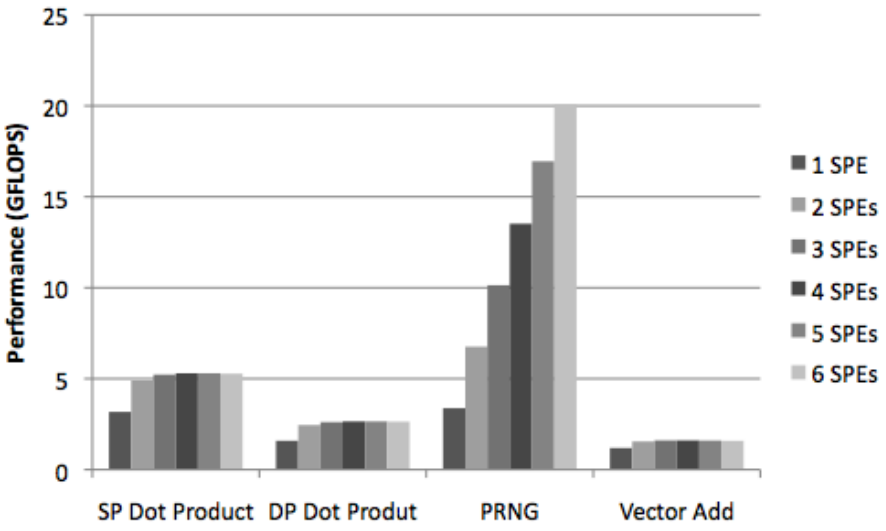


Figure 6.5: Benchmark performance in GFLOPS.

memory bound. Vector addition is a very simple operation, and the high communication complexity is the bottleneck.

Figure 6.4 shows the memory bandwidth used by the benchmarks as a fraction of the experimental limit. As expected, the dot product benchmark and the vector add benchmark both use all of the available bandwidth for 1-6 SPEs, while the PRNG benchmark fails to use the system's entire bandwidth even with 6 SPEs. In all of these cases, increasing the degree of parallelism on-chip will not result in improved performance, and would lead to decreased power efficiency.

Figure 6.5 shows the raw performance of the four benchmarks in gigaflops for 1-6 SPEs. These results are a combination of the computational and communication complexity of each benchmark. The problem with the Cell processor is most evident here, especially with the vector addition benchmark. Because every single operation requires three data elements (two inputs and one output), memory bandwidth limits the peak performance to

only 1.6 GFLOPS out of the total 153.6 GFLOPS theoretically available; the benchmark only uses 1% of the processor's available computing power.

7 Conclusion

Designing software or hardware for a CMP is always difficult. It is important, however, that the design doesn't ignore the primary objective of increased processor performance at a lower cost. When a feature of the hardware's architecture prevents this objective from being realized, it is necessary to reexamine our understanding of the interaction of hardware and software.

The design of the memory subsystem of a CMP is a great challenge with myriad problems. For shared-memory architectures, cache coherence is a big issue, and hardware and labor have been expended to ensure that all levels of cache have valid data as necessary. This thesis, however, considered a distributed-memory architecture with streaming processors, and focused on the problems presented by such an architecture.

We can see why off-chip memory bandwidth is so often the bottleneck on the Cell/B.E. The maximum performance of the 6-SPEs PlayStation 3 is 153.6 GFLOPS. Each pair of floating-point operations can use up to four 4-byte data elements (inputs and outputs), making the maximum computational throughput of the system 1,228.8 GB/s of data. The off-chip memory bandwidth of 25.6 GB/s is just over 2% of this value. Although 1,228.8 GB/s is an unreachable upper limit which assumes absolutely no data reuse and the issue of one multiply-add instruction every cycle, it is clear that a large gap exists between the computational power of the Cell processor and the bandwidth supported by memory.

More recent and future microprocessor architectures have sought to remove this off-

chip memory bottleneck. NVIDIA's Tesla and AMD's FireStream, both designed for data-intensive computing, offer over 100 GB/s of peak memory bandwidth each, several times greater than the Cell's [8, 15]. Sun's UltraSPARC T2 has over 60 GB/s in memory bandwidth and a 4MB L2 cache, which will reduce off-chip memory demands [16]. Intel is hoping to develop 3D memory to give its Teraflops Research Chip memory bandwidth in the hundreds of gigabytes per second [7].

Increasing power efficiency is also important to hardware developers: Intel and Sun are both developing high-performance computing processors that use under 100 W [7, 17]. As we have seen, existing CMPs can already offer increased power efficiency over traditional processors.

Our experiments showed that the Cell processor is energy efficient, capable of reducing the total energy use of an application by over 61% compared to the same application on a traditional x86-64 processor. However, memory bandwidth is a significant limitation for embarrassingly parallel data-intensive applications, limiting speedup to under 28% of the linear speedup target on 6 SPEs.

7.1 Future Work

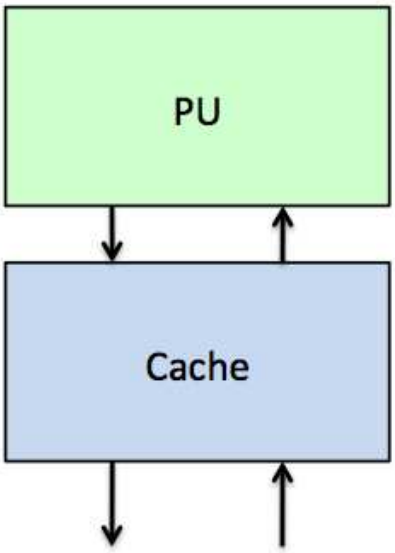
As memory management is one of the most challenging aspects of CMP programming, there is a need to develop methods for the automation of data staging. A high-level memory management API could take OpenMP as a model to maximizing performance while reducing the difficulty of programming. Making such an API architecture-independent would also be important, and would greatly serve the scientific computing community.

Future extensions of this work may examine other architectures with the objectives presented here in mind. As more multicore and vector processors become commercially available, the issues of the memory subsystem of all may be studied to create a thorough

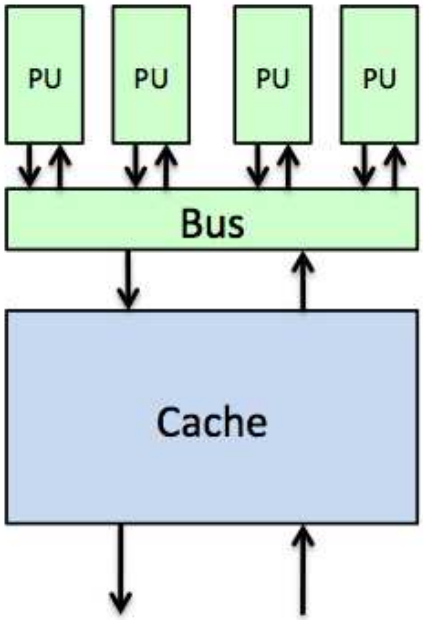
mapping of the relationships between memory latency, bus width, processor speed, and interconnect architecture.

Appendix A. Chip Multiprocessors

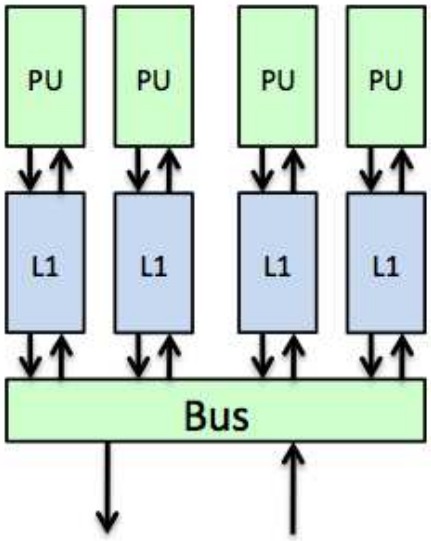
Basic schematics of a uniprocessor and several CMPs are shown in Figure A.1. Buses, on-chip memory, and processing units are included. For a comparison, see Figure 3.2 for a schematic of the Cell/B.E. processor.



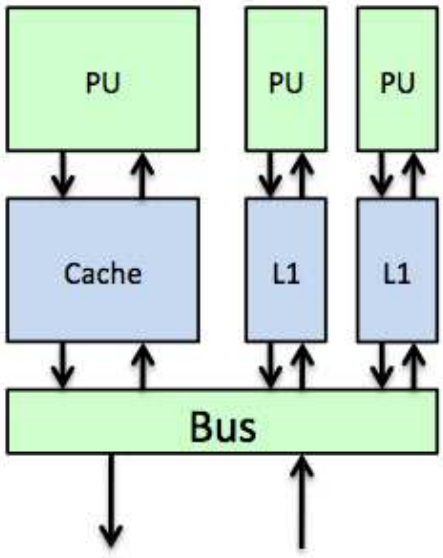
(a) A traditional uniprocessor, with a L1/2 cache and a single processing unit (PU).



(b) A CMP with shared memory. The addressable memory of each PU is the same; caches may be distinct but are kept coherent.



(c) A CMP with distributed memory. Each PU has a separate addressable memory, but they may share I/O buses.



(d) A heterogeneous, distributed-memory CMP, similar to the Cell architecture seen in figure 3.2. Here, not all processing elements have the same capacities.

Figure A.1: Basic chip multiprocessors

Bibliography

[1] Carmelo Acosta, Francisco J. Cazorla, Alex Ramirez, and Mateo Valero. Core to memory interconnection implications for forthcoming on-chip multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*, Phoenix, Arizona, February 2007.

[2] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA*, pages 248–259, 2000.

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, New York, NY, USA, 2008. ACM.

[4] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. *Micro, IEEE*, 25(3):32–45, May-June 2005.

[5] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation: a performance view. *IBM J. Res. Dev.*, 51(5):559–572, 2007.

[6] IBM Corporation. Cell Broadband Engine. `http://domino.research.ibm.com/comm/research_projects.nsf/pages/multicore.CellBE.html`.

[7] Intel Corporation. Advancing Multi-Core Technology into the Tera-scale Era. `http://techresearch.intel.com/articles/Tera-Scale/1449.htm`.

[8] NVIDIA Corporation. What is GPU Computing? `http://www.nvidia.com/object/GPU_Computing.html`, 2009.

[9] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the CELL processor. In *PACT '05: Proceedings of the 14th International Conference on*

Parallel Architectures and Compilation Techniques, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[10] B. Flachs, S. Asano, S. H. Dhong, H. P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. S. Liberty, B. Michael, H.-J. Oh, S. M. Mueller, O. Takahashi, K. Hirairi, A. Kawasumii, H. Murakami, H. Noro, S. Onishi, J. Pille, J. Silberman, S. Yong, A. Hatakeyama, Y. Watanabe, N. Yano, D. A. Brokenshire, M. Peyravian, V. To, and E. Iwata. Microarchitecture and implementation of the synergistic processor in 65-nm and 90-nm SOI. *IBM J. Res. Dev.*, 51(5):529–543, 2007.

[11] B. Flachs, S. Asano, S.H. Dhong, P. Hotstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S.M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A streaming processing unit for a CELL processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 134–135 Vol. 1, February 2005.

[12] K. A. Gallivan, Michael T. Heath, Esmond Ng, James M. Ortega, Barry W. Peyton, R. J. Plemmons, Charles H. Romine, and Robert G. Voigt. *Parallel Algorithms for Matrix Computations*. SIAM, 1990.

[13] Michael Gschwind. Chip multiprocessing and the Cell Broadband Engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, New York, NY, USA, 2006. ACM.

[14] IBM Corporation. *Cell Broadband Engine Programming Handbook Including PowerXCell 8i*, May 2008.

[15] AMD Inc. AMD FireStream 9250. `http://ati.amd.com/technology/streamcomputing/product_firestream_9270.html`, 2009.

[16] Sun Microsystems Inc. UltraSPARC T2 Processor: The world's first true system on a chip. Datasheet. `http://www.sun.com/processors/UltraSPARC-T2/datasheet.pdf`, 2007.

[17] Sun Microsystems Inc. UltraSPARC T1: Overview. `http://www.sun.com/processors/UltraSPARC-T1/`, 2009.

[18] D. Jimenez-Gonzalez, X. Martorell, and A. Ramirez. Performance analysis of Cell Broadband Engine for high memory bandwidth applications. *Performance Analysis of Systems and Software, IEEE International Symmposium on*, 0:210–219, 2007.

[19] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[20] Pierre L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Math. Comput.*, 68(225):249–260, 1999.

[21] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. Comparing memory systems for chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):358–368, 2007.

[22] Yingmin Li, David Brooks, Zhigang Hu, and Kevin Skadron. Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. *High-Performance Computer Architecture, International Symposium on*, 0:71–82, 2005.

[23] J. D. C. Little. A proof of the queuing formula: $L = \lambda W$. *Operations Research*, 9(3), 1961.

[24] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *10th IEEE Symposium on High-Performance Computer Architecture*, pages 176–185, February 2004.

[25] B. A. Nayfeh and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, Sep 1997.

[26] S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *Commun. ACM*, 31(10):1192–1201, 1988.

[27] John Shalf, Bill Tschudi, Stephen Elbert, Rob Pennington, Andres Marques, Tim McCann, and Tahir Cader. Power, cooling, and energy consumption for the petascale and beyond. In *Supercomputing 2007*, 2007.

[28] Harold S. Stone. *High-Performance Computer Architecture*, chapter 6, pages 304–309. Addison-Wesley, 1990.

[29] Li Zhao, Ravi Iyer, Srihari Makineni, Ramesh Illikkal, Jaideep Moses, and Don Newell. Constraint-aware large-scale CMP cache design. In *Lecture Notes in Computer Science*, volume 4873. Springer, Berlin / Heidelberg, 2007.

[30] Li Zhao, Ravi Iyer, Srihari Makineni, Jaideep Moses, Ramesh Illikkal, and Don Newell. Performance, area and bandwidth implications on large-scale CMP cache design. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*, Phoenix, Arizona, February 2007.